

React

Introduction to Redux Saga

by Aryan Sajan Kulathinal

What is React Saga?

- 1 Definition: Redux-Saga is a middleware library specifically designed to handle side effects in Redux-managed React applications.
- 2 Purpose: Manages asynchronous actions such as: Making API calls, Handling WebSocket connections, Managing complex asynchronous workflows
- 3 Foundation: Built on ES6 generator functions, which allow functions to pause and resume execution, making it ideal for non-blocking asynchronous operations.
- 4 Functionality: Facilitates async communication between the Redux store and external resources (e.g., APIs, local storage). Efficiently handles input/output services and async tasks like HTTP requests.
- 5 Middleware Example: Redux-Saga is a type of Redux middleware, alongside others like Redux Thunk, for handling custom dispatch behaviors and async flows.



Redux

- Definition:** Redux is a centralized state management library that allows all components in an application to access shared data efficiently.

- Challenges with Prop Drilling:**

- Managing data through props becomes unwieldy in large applications with many components.

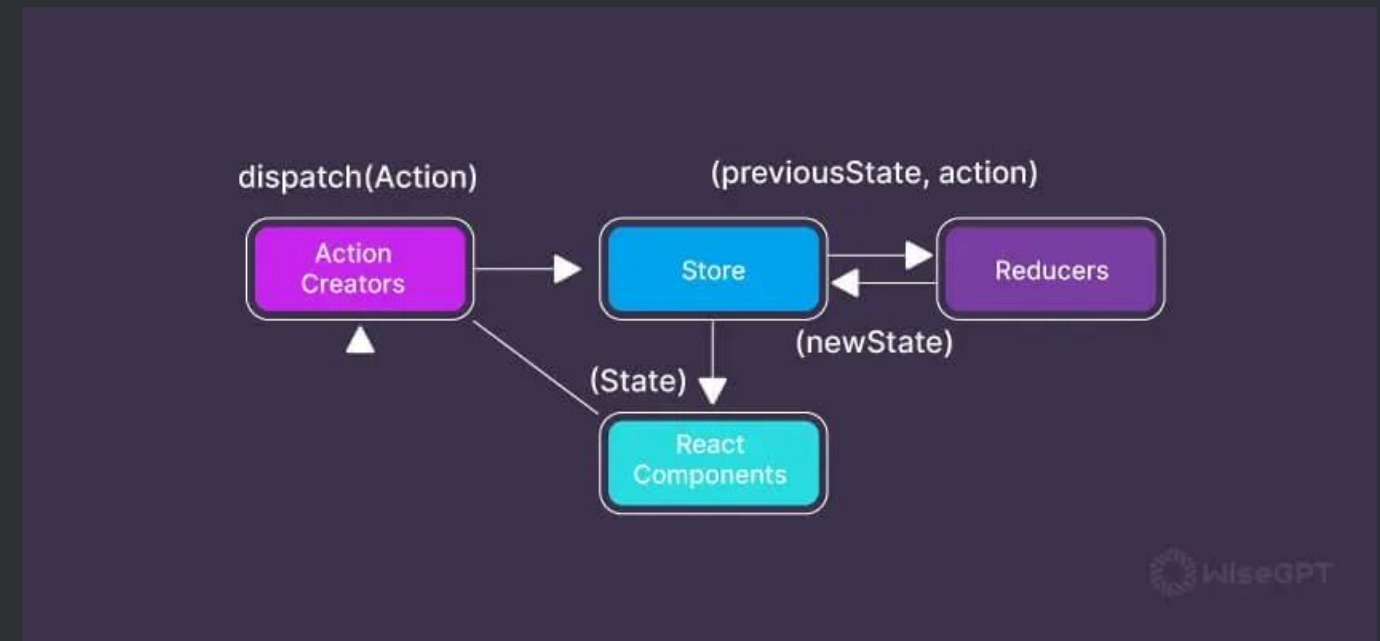
- Frequent changes to any component can trigger re-renders in all connected components, negatively impacting performance and user experience.

- Solution with Redux:**

- Redux provides a centralized store for managing application state.

- By using Redux, components can access the data they need directly from the central store, reducing the need for prop drilling and improving optimization.

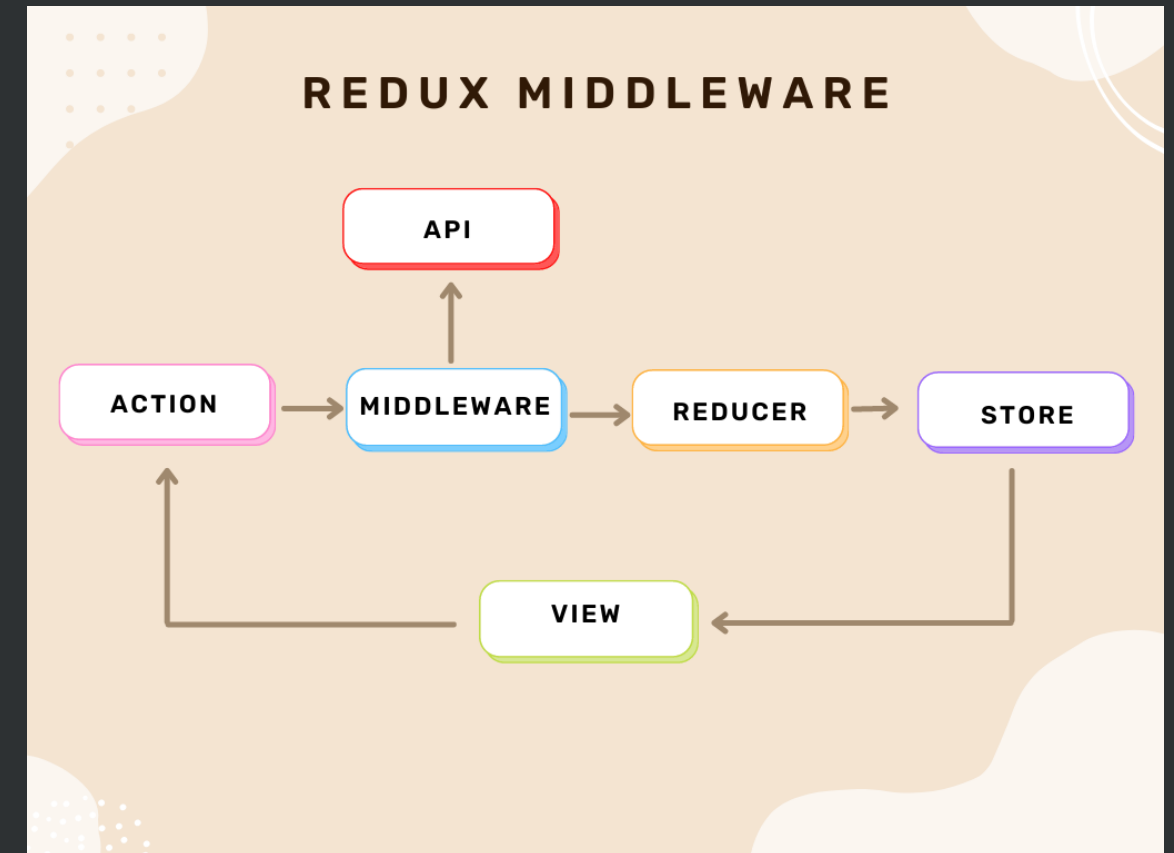
- This approach leads to more efficient state management and better performance in larger applications.

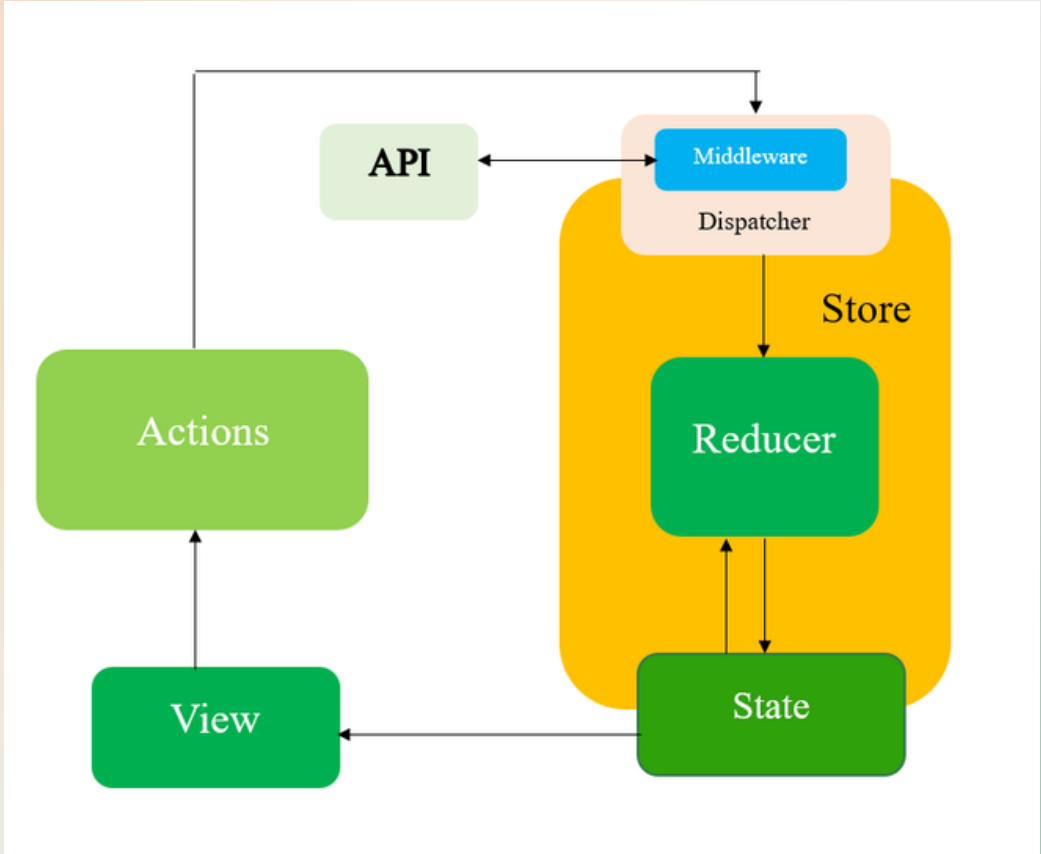


Middleware

Middleware in Redux

- **Definition:** Middleware in Redux is a way to extend the functionality of the Redux store, allowing for additional features to be integrated into the application.
- **Functionality:**
 - Acts as an intermediary between the dispatching of an action and its arrival at the reducer.
 - Enables third-party extensions and custom behaviors during the dispatch process.
- **Use Cases:**
 - **Crash Reporting:** Capture errors and send reports.
 - **Logging:** Track dispatched actions and state changes for debugging.
 - **Asynchronous Operations:** Manage async tasks such as API calls and delayed actions.
- **Customization:**
 - Enhancers are used to modify the dispatch function, but middleware allows for deeper customization.
 - Similar to middleware in other frameworks (e.g., Express), Redux middleware customizes specific behaviors within an application.





Store

Definition: The centralized place where the application state lives in Redux.**Role:** Holds the current state, provides dispatch for actions, and allows subscription to state updates.**Relationship:** Coordinates between all elements, updating the state and notifying the view to re-render based on changes.

Redux WorkFlow with Middleware

State

Definition: The single source of truth in Redux, representing the entire application's data.**Role:** Centralized in the Redux store, it can be accessed by any component.**Relationship:** When an action changes the state, the view re-renders based on the new state.

Actions

Definition: Plain JavaScript objects describing an event that has occurred (e.g., fetching data, updating an item).**Role:** Contains a type field (indicating the event type) and, often, a payload (additional data for the reducer).**Relationship:** Actions are dispatched to the store, initiating the flow to update the state.

API

Definition: External services or endpoints accessed for data (e.g., fetching product lists, saving user information).**Role:** Provides data or processing services for the application, typically accessed through asynchronous actions.**Relationship:** Middleware (e.g., Redux Saga) makes API calls when an action is dispatched, fetching or updating data before it flows to reducers

View

Definition: The UI components or pages in an application, often represented by React components.**Role:** Reads data from the state and displays it to users; also triggers actions in response to user interactions.**Relationship:** Re-renders whenever the state updates, providing the latest data to users.

Dispatcher

Definition: A function (dispatch) that sends actions to the Redux store.**Role:** Connects actions to the Redux store, triggering state changes.**Relationship:** When the view calls dispatch, it sends the action to middleware (if any) and then to the reducers

Reducer

Definition: Pure functions that take the current state and an action, returning a new state based on the action's type and payload.**Role:** Determines how the state should change in response to actions.**Relationship:** Receives actions after they pass through any middleware, applies the action logic, and returns an updated state to the store.

Why use Redux Saga?

Simplification of Async Handling

Redux Saga simplifies the management of complex asynchronous flows in applications, making code more manageable and readable.

Testability

Sagas can be easily tested in isolation, making your application more robust and maintainable. The design of Redux Saga allows for easy testing of asynchronous logic through its generator functions, enabling step-by-step execution and simulation of side effects.

Declarative Syntax/Readability

Sagas use a declarative approach, making your application's business logic more readable and understandable.

Error Handling

Provides robust mechanisms for handling failures in async operations, improving application reliability.

Scalability

Sagas help manage complex workflows and scale your application as it grows in complexity.

Sagas And their Functions

1

Dispatching Actions

Sagas listen for actions dispatched from your components and handle the associated side effects.

2

Asynchronous Flows

Sagas use generator functions to describe complex asynchronous flows, like data fetching and error handling.

3

Coordinating Effects

Sagas can coordinate multiple effects, like race conditions and parallel executions, to manage complex workflows.

Sagas in Redux-Saga are generator functions that manage asynchronous side effects in a Redux application, like making API calls, handling complex business logic, or managing non-blocking operations such as animations or timeout events. Redux-Saga uses these generator functions to “watch” for specific actions and then execute related asynchronous code in a controlled, predictable way.

Core Concepts in Redux Saga

1. Effects: The building blocks for handling asynchronous tasks. Key effects include:

❖ **call:**

- ❖ Used to invoke asynchronous functions or methods.
- ❖ Can take parameters for the function being called, allowing flexibility.

❖ **put:**

- ❖ Dispatches actions to the Redux store.
- ❖ Useful for sending updates back to the store after an async operation is complete.

❖ **take:**

- ❖ Pauses the saga execution until a specific action is dispatched.
- ❖ Useful for waiting for user interactions or specific events before proceeding.

❖ **fork:**

- ❖ Creates a non-blocking task that runs independently.
- ❖ Allows concurrent execution of tasks without waiting for them to complete.

❖ **race:**

- ❖ Runs multiple effects concurrently and cancels all except the first one that completes.
- ❖ Useful for managing timeouts or handling scenarios where only one response is needed.

2. Advanced Effects: Enhance control over asynchronous operations.

❖ **takeLatest:**

- ❖ Ignores all previous actions of the same type if a new action is dispatched before the previous one finishes.
- ❖ Ensures that only the latest request is processed, ideal for scenarios like user input where only the most recent action matters.

❖ **takeEvery:**

- ❖ Allows handling every dispatched action of a specific type.
- ❖ Useful for scenarios where each action needs to be processed independently, like logging or analytics.

❖ **debounce:**

- ❖ Delays execution until a specified time has passed since the last time it was invoked.
- ❖ Useful for scenarios like search inputs where you want to reduce the number of API calls based on user input frequency.

❖ **throttle:**

- ❖ Limits the number of times a function can be called over a specified period.
- ❖ Useful for preventing performance issues caused by rapid user interactions, like button clicks.

Setting up a React Project using React Saga....

Tutorial: Building a Random Number Generator App with React, TypeScript, and Redux-Saga

Step 1: Create a React Application

1. Open your terminal and create a new React application using Create React App with TypeScript template:

```
bash
```

[Copy code](#)

```
npx create-react-app random-number-generator --template typescript
```

Step 2: Install Required Packages

2. Navigate to your project directory:


```
bash
```

[Copy code](#)

```
cd random-number-generator
```

3. Install Redux, React-Redux, Redux-Saga, and Redux Toolkit:

bash

 Copy code

```
npm install @reduxjs/toolkit react-redux redux-saga
```

Step 3: Project Structure

- Create the following folders and files:
 - `src/store/index.ts`
 - `src/store/actions.ts`
 - `src/store/reducer.ts`
 - `src/store/sagas.ts`
 - `src/components/RandomNumberGenerator.tsx`

Step 4: Create Actions

- Open `src/store/actions.ts` and define  action types and action creators:

typescript

 Copy code

```
// src/store/actions.ts
```

```
export const GENERATE_RANDOM_NUMBER_REQUEST = 'GENERATE_RANDOM_NUMBER_REQUEST';
```

```
export const GENERATE_RANDOM_NUMBER_SUCCESS = 'GENERATE_RANDOM_NUMBER_SUCCESS';
```

```
export const GENERATE_RANDOM_NUMBER_FAILURE = 'GENERATE_RANDOM_NUMBER_FAILURE';
```

```
export const generateRandomNumberRequest = () => ({  
  type: GENERATE_RANDOM_NUMBER_REQUEST,  
});
```

```
export const generateRandomNumberSuccess = (number: number) => ({  
  type: GENERATE_RANDOM_NUMBER_SUCCESS,  
  payload: number,  
});
```

```
export const generateRandomNumberFailure = (error: string) => ({  
  type: GENERATE_RANDOM_NUMBER_FAILURE,  
  payload: error,  
});
```

Step 5: Create Reducer

- Open `src/store/reducer.ts` and implement the reducer to handle the actions:

```
typescript Copy code

// src/store/reducer.ts
import { AnyAction } from 'redux';
import {
  GENERATE_RANDOM_NUMBER_SUCCESS,
  GENERATE_RANDOM_NUMBER_FAILURE,
} from './actions';

interface State {
  randomNumber: number | null;
  error: string | null;
}

const initialState: State = {
  randomNumber: null,
  error: null,
};

const reducer = (state = initialState, action: AnyAction): State => {
  switch (action.type) {
    case GENERATE_RANDOM_NUMBER_SUCCESS:
      return {
        ...state,
        randomNumber: action.payload,
        error: null,
      };
    case GENERATE_RANDOM_NUMBER_FAILURE:
      return {
        ...state,
        randomNumber: null,
        error: action.payload,
      };
    default:
      return state;
  }
};

export default reducer;
```

Step 6: Create Sagas

- Open `src/store/sagas.ts` and implement the saga to handle asynchronous operations:

typescript

 Copy code

```
// src/store/sagas.ts
import { call, put, takeEvery } from 'redux-saga/effects';
import {
  GENERATE_RANDOM_NUMBER_REQUEST,
  generateRandomNumberSuccess,
  generateRandomNumberFailure,
} from './actions';


function* generateRandomNumber() {
  try {
    // Simulate a random number generation
    const number = Math.floor(Math.random() * 100) + 1;
    yield put(generateRandomNumberSuccess(number));
  } catch (error) {
    yield put(generateRandomNumberFailure('Failed to generate a random number'));
  }
}

export function* watchGenerateRandomNumber() {
  yield takeEvery(GENERATE_RANDOM_NUMBER_REQUEST, generateRandomNumber);
}
```

Step 7: Set Up Redux Store

- Open `src/store/index.ts` and set up the Redux store with middleware:

typescript

 Copy code

```
// src/store/index.ts
import { configureStore } from '@reduxjs/toolkit';
import createSagaMiddleware from 'redux-saga';
import reducer from './reducer';
import { watchGenerateRandomNumber } from './sagas';

const sagaMiddleware = createSagaMiddleware();

const store = configureStore({
  reducer,
  middleware: (getDefaultMiddleware) => getDefaultMiddleware().concat(sagaMiddleware),
});


sagaMiddleware.run(watchGenerateRandomNumber);

export default store;
```


Step 8: Create the Random Number Generator Component

- Open `src/components/RandomNumberGenerator.tsx` and implement the component:

typescript

 Copy code

```
// src/components/RandomNumberGenerator.tsx
import React from 'react';
import { useDispatch, useSelector } from 'react-redux';
import { generateRandomNumberRequest } from '../store/actions';

const RandomNumberGenerator: React.FC = () => {
  const dispatch = useDispatch();
  const randomNumber = useSelector((state: any) => state.randomNumber);
  const error = useSelector((state: any) => state.error);

  const handleGenerateRandomNumber = () => {
    dispatch(generateRandomNumberRequest());
  };

  return (
    <div className="app-container">
      <h1>Random Number Generator</h1>
      <button className="generate-button" onClick={handleGenerateRandomNumber}>
        Generate Random Number
      </button>
      {randomNumber !== null && (
        <div className="result">
          <h2>Random Number: {randomNumber}</h2>
        </div>
      )}
      {error && <div className="error">{error}</div>}
    </div>
  );
};

export default RandomNumberGenerator;
```

Step 9: Update the App Component

- Open `src/App.tsx` and modify it to include the `RandomNumberGenerator` component:

typescript

 Copy code

```
// src/App.tsx
import React from 'react';
import RandomNumberGenerator from '../components/RandomNumberGenerator';
import { Provider } from 'react-redux';
import store from './store';
import './App.css';

const App: React.FC = () => {
  return (
    <Provider store={store}>
      <div className="App">
        <RandomNumberGenerator />
      </div>
    </Provider>
  );
};

export default App;
```

Step 10: Add Basic Styling

1. Open `src/App.css` and add the following styles:

```
css Copy code

/* src/App.css */
.app-container {
  text-align: center;
  margin: 50px;
  padding: 20px;
  border: 1px solid #ccc;
  border-radius: 5px;
  background-color: #f8f8f8;
  box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);
}

h1 {
  font-size: 24px;
  margin-bottom: 20px;
}

.generate-button {
  background-color: #007bff;
  color: #fff;
  padding: 10px 20px;
  font-size: 16px;
  border: none;
  border-radius: 5px;
  cursor: pointer;
}

.generate-button:hover {
  background-color: #0056b3;
}


.result {
  margin-top: 20px;
  font-size: 20px;
  color: #333;
}

.error {
  color: red;
  margin-top: 10px;
}
```

Step 11: Run the Application

- Use the following command to start your application:

```
bash
```

 Copy code

```
npm start
```

- Open your browser and navigate to `http://localhost:3000/`. You should see the Random Number Generator app.

Final Output

Your app will display a button that, when clicked, generates a random number between 1 and 100 asynchronously using Redux-Saga and displays it on the screen.

Comparison B/w Thunks and Saga

| Feature | Redux Thunk | Redux Saga |
|----------------|--------------------------------------|--------------------------------------|
| Basic Approach | Functions as actions | Generator functions as sagas |
| Concurrency | Limited | Advanced control (takeLatest, race) |
| Ease of Use | Simple setup | More complex, steep learning curve |
| Error Handling | Basic try-catch | Built-in handling across workflows |
| Best for | Small/medium apps, basic async needs | Large apps, complex async workflows |
| Performance | Lightweight | More overhead, better for large apps |



Thank you

This presentation has provided an overview of React Saga and how it can simplify the management of side effects in your React applications. By using Sagas, you can build more scalable, maintainable, and testable applications.